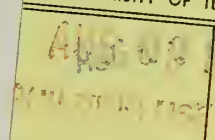


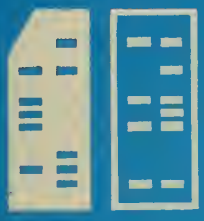
UIUCDCS-R-74-655

CLUSTERING BY CLIQUE GENERATION

by

Chih-Meng Cheng

June, 1974



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-74-655

ERATION

June, 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, June, 1974

UIUCDCS-R-74-655

CLUSTERING BY CLIQUE GENERATION

by

Chih-Meng Cheng

June, 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, June, 1974

ACKNOWLEDGEMENT

I wish to thank my advisor, Professor D. S. Watanabe, for his constant guidance, patience, and encouragement, in the preparation of this thesis.

Financial support from the Department of Computer Science of the University of Illinois is also gratefully acknowledged.

TABLE OF CONTENTS

1. INTRODUCTION

Given a set of objects each described by a vector of characteristics, a clustering technique groups those objects with similar characteristics together into subsets called clusters. The similarity criterion uses an appropriate distance function measuring the distance between objects which varies with the interpretation of the characteristic vector for the set.

Clustering techniques are useful in many areas. For example, they can be used in medicine to identify new diseases and to refine existing disease categories, in biology to develop taxonomies for plants and animals, and in archaeology to classify artifacts with respect to period and style.

There is no general method which always yields useful clusters for an arbitrary set of objects. Usually different techniques are tried, and often relevant clusters can be obtained through comparison of the results. Two of the most popular and effective techniques are the single-link method [5] and clique generation.

In both methods, the set of objects is interpreted as an undirected graph. For a given distance function, we can define a threshold δ such that if the distance between two objects is less than δ, then the two objects are said to be similar. Using this concept, the set of objects can be interpreted as a graph where nodes represent objects and edges join nodes representing similar objects.

The single-link method is the simplest and oldest technique. In this method a cluster is defined to be a connected component of the graph. A connected component is a subgraph in which each pair of nodes is joined by a path or sequence of edges. For objects which form distinct disconnected

clusters, the single-link method often yields good results (see Figure 1a), and the simplicity of the method allows it to be applied efficiently to large sets of objects. However, the obvious disadvantage of this method is the chaining effect, where pairs of nodes may be joined by a path but the distance between the objects they represent is large (see Figure 1b).

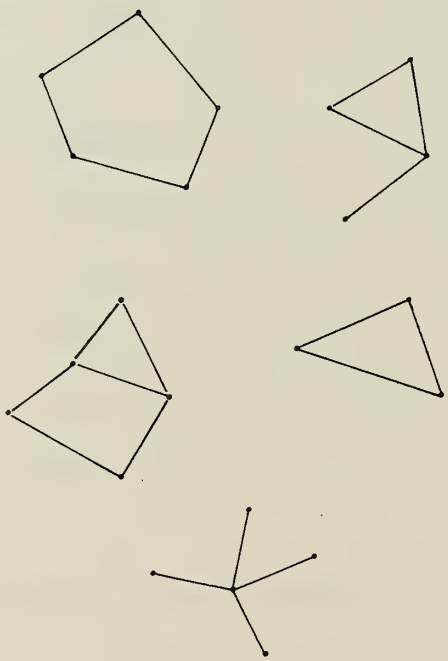


Figure 1a. Single-link clusters

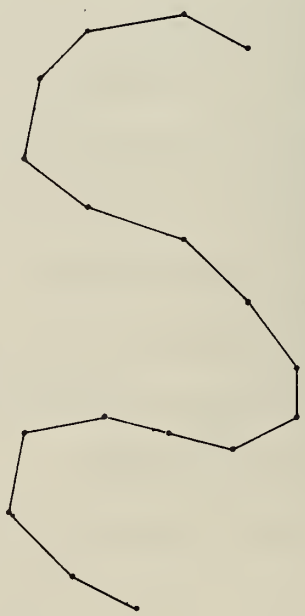


Figure 1b. Chaining effect

In clique generation a cluster is defined to be a maximal complete subgraph or clique of the graph. A complete subgraph is one in which every node in the subgraph is adjacent to every other node in the subgraph. A maximal complete subgraph is a complete subgraph which is not properly contained in any other

complete subgraph. The problem of finding all the cliques in an arbitrary graph is well known, and many algorithms have been proposed. The earliest were developed by Bonner and Bierstone [1], but the most efficient algorithm available was devised by Bron and Kerbosch [2]. Unfortunately finding cliques is much more difficult than finding connected components. In fact the problem of finding all the cliques in an arbitrary graph is polynomial-complete, and hence is equivalent in difficulty to the notoriously difficult "traveling-salesman" problem.

These two methods illustrate how tightly clusters can be defined by lying at the extremes of any reasonable scale of tightness. Cliques contain more information about the structure of a graph than connected components, and although they are often too tight to be used as clusters, they can form the nuclei of useful clusters. Hence we will restrict our attention to clique generation.

Chapter 2 presents a detailed study of the Bron-Kerbosch algorithm. The algorithm is described, analyzed, and shown to be near optimal. Chapter 3 discusses the efficient implementation of the algorithm, describes an efficient new implementation, and presents numerical results demonstrating the superiority of the new implementation over previous ones.

2. THE BRON-KERBOSCH ALGORITHM

2.1 Analysis

Mulligan [4] studied the algorithms of Bonner, Bierstone, and Bron and Kerbosch in detail. His tests showed that the Bron-Kerbosch algorithm is fastest. To generate n cliques, it required time of $O(n^{1+\varepsilon})$, where ε is a small positive quantity, while the other algorithms required time of $O(n^2)$. Unfortunately the number of cliques in a graph generally is an exponential function of the number of nodes in the graph. Hence the speed of an algorithm is crucial.

Bron and Kerbosch's presentation of their algorithm is not well motivated. Hence we will attempt to state clearly the motivation behind their algorithm and to explain why it works so well.

Figure 2 illustrates how clique generation can be applied to a simple data set. Figure 2a presents the adjacency matrix A for the six objects in the set. Here a_{ij} is 1 if and only if object i is similar to object j; otherwise a_{ij} is 0. Figure 2b presents a graph G representing the data set. The nodes in G correspond to the objects in the set, and an edge connects any two nodes in G corresponding to similar objects. Figure 2c presents a graph Γ summarizing the possible ways of generating the cliques of G, starting with the empty set, ϕ. Each node in Γ is a complete subgraph, and each edge from a node α in level ℓ to a node β in level $\ell + 1$ is labeled with the node of G added to α to form β. A clique is generated by traversing a path or sequence of edges which terminates in a clique.

Obviously one way to generate all the cliques is to visit every node and traverse all the paths in Γ. This approach is time-consuming and wasteful

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	0	1	1
4	0	1	0	0	0	0
5	0	0	1	0	0	1
6	0	0	1	0	1	0

Figure 2a. Adjacency Matrix A

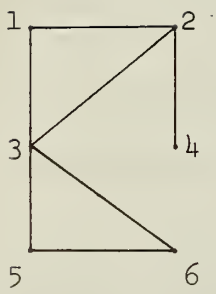


Figure 2b. Graph G

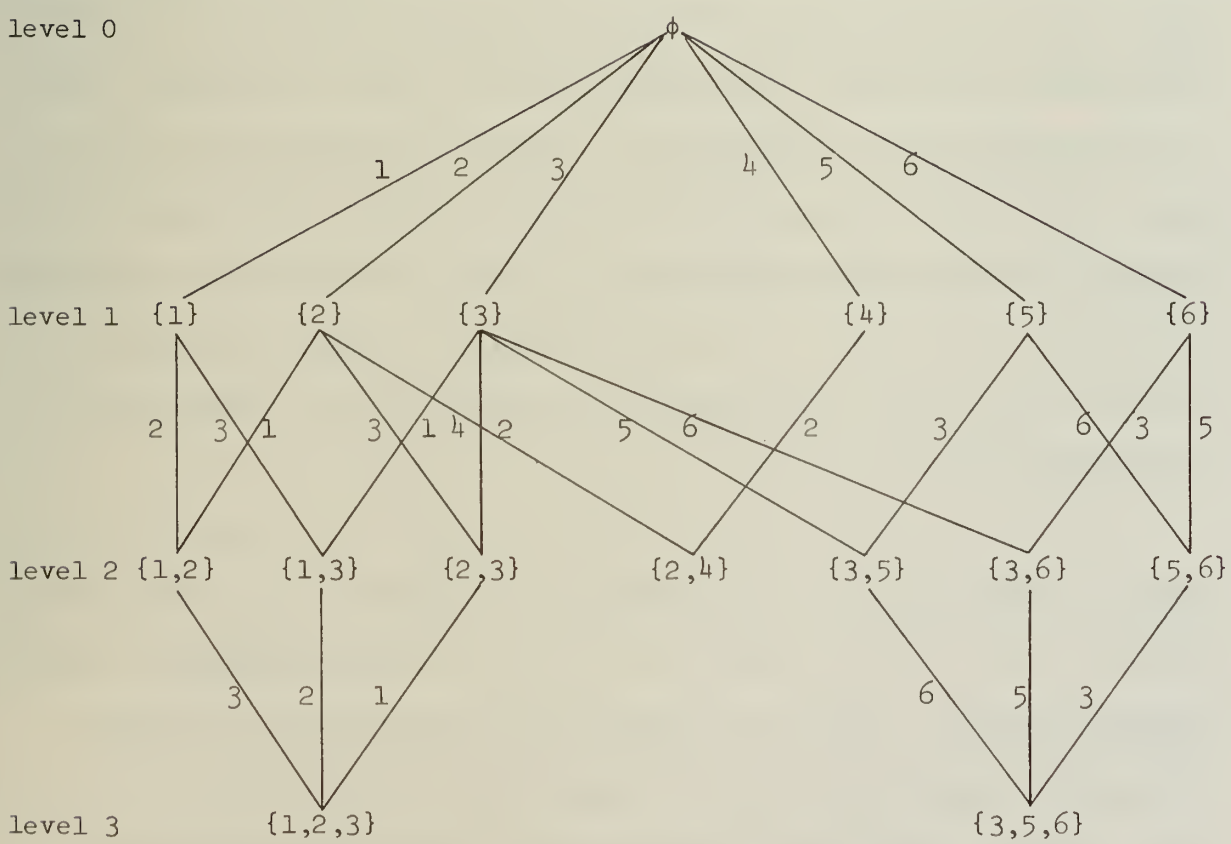


Figure 2c. Clique Generation Graph Γ

because most of the paths lead to cliques already generated. Most early algorithms perform an ordered traversal of Γ, but this is still wasteful because subsets of already generated cliques are repeatedly constructed. Bron and Kerbosch used a cleverer approach and were able to eliminate certain paths by applying the ideas formalized in the following lemmas.

Lemma 1. Suppose the paths from node α in Γ beginning with the edge labeled with node a of G have been explored so that all cliques containing $\alpha \cup \{a\}$ have been generated. Then only those paths from α beginning with edges labeled with nodes of G not adjacent to a need be explored.
Proof. Let C be any clique generated by exploring a path from α beginning with an edge labeled with a node adjacent to a. Then it must either contain or not contain nodes not adjacent to a. Suppose it contains such a node, say b. Then clearly it can be generated by exploring a path from α beginning with the edge labeled with node b which is not adjacent to a. Suppose it contains no such nodes. It obviously contains α, and it must contain a since all its other nodes by assumption are adjacent to a. Therefore it has already been generated. Q.E.D.

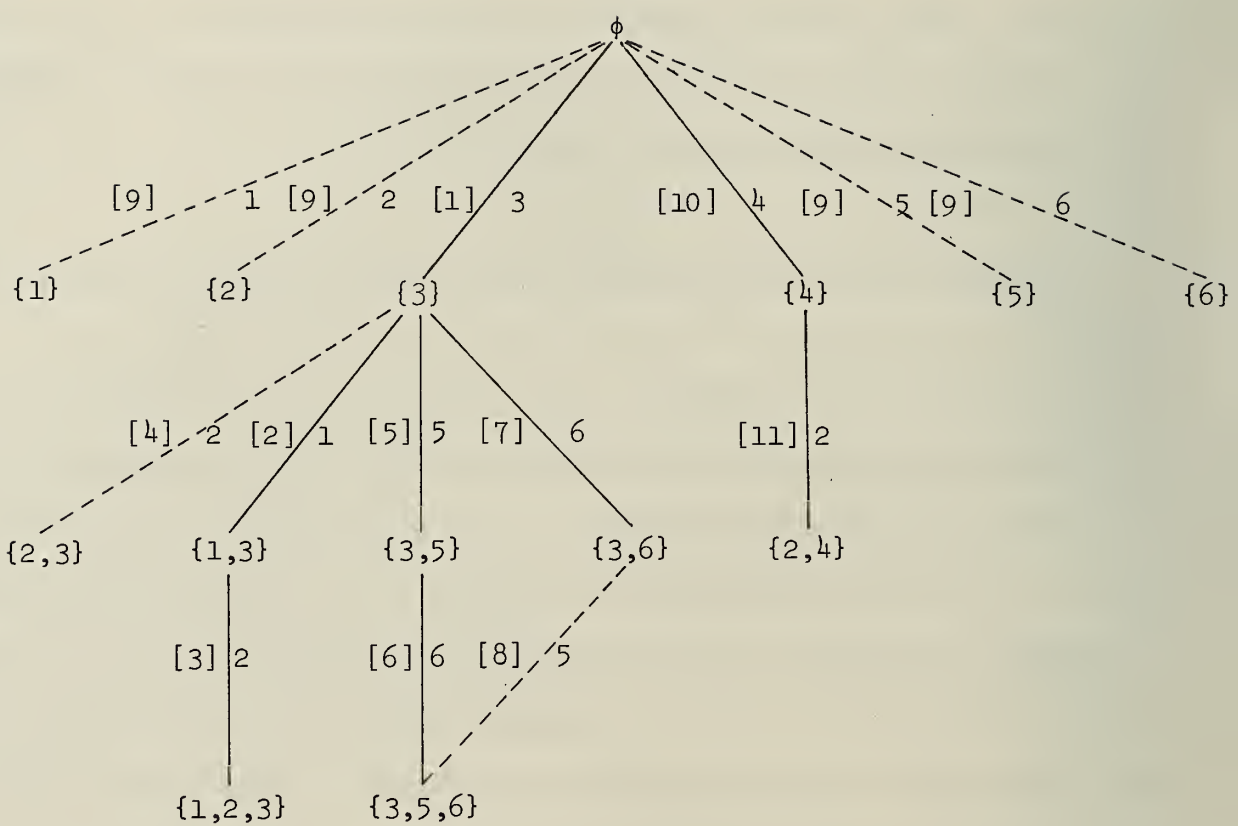
Lemma 2. Suppose the paths from node α in Γ beginning with the edge labeled with node a of G have been explored so that all cliques containing $\alpha \cup \{a\}$ have been generated. Then at any node β of Γ which properly contains α, those paths beginning with an edge labeled with a can be ignored.
Proof. Suppose a path from β beginning with an edge labeled with a is

explored. It must clearly terminate in a clique containing $\alpha \cup \{a\}$. But by assumption, all the cliques containing $\alpha \cup \{a\}$ have already been generated, and hence the path can be ignored. Q.E.D.

The Bron-Kerbosch algorithm is recursive; upon arriving at a node in level ℓ, the algorithm calls itself to explore the levels higher than ℓ. Lemma 1 is applied at each level. Upon first arriving at node α in Γ at level ℓ, the algorithm selects a node of G called FIXP that is adjacent to the most nodes adjacent to the partially constructed clique α, moves to the node $\alpha \cup \{FIXP\}$, and calls itself to construct all the cliques containing $\alpha \cup \{FIXP\}$. This choice of FIXP eliminates the maximum number of paths from α. Upon returning to node α, the algorithm chooses a node of G called SEL that is adjacent to the nodes in α but not adjacent to FIXP, moves to the node $\alpha \cup \{SEL\}$, calls itself to construct all the cliques containing $\alpha \cup \{SEL\}$, and repeats this procedure for all such nodes. This process is illustrated for the graph G of Figure 2b in Figure 3.

Lemma 1 cannot be used to eliminate all redundant edges. Note, for example, that the edge labeled 6 from node {3} in Figure 3 is traversed although it leads to the clique {3,5,6}, previously generated. However, lemma 2 can be used to eliminate the edge labeled 5 from node {3,6}, and the clique is not regenerated.

Lemma 1 is used only once at each level for the node FIXP. Conceivably it could be applied repeatedly for every node SEL at each level. However, if it is used to eliminate edges labeled with nodes adjacent to SEL, some



[1] Select 3 as FIXP at level 0, and move to {3}.

[2] Select 1 as FIXP at level 1, and move to {1,3}.

[3] Select 2 as FIXP at level 2, move to clique {1,2,3}, and back up to {3}.

[4] Ignore 2 because 2 is adjacent to 1, FIXP at level 1.

[5] Select 5 as SEL at level 1 and move to {3,5}.

[6] Select 6 as FIXP at level 2, move to clique {3,5,6}, and back up to {3}.

[7] Select 6 as SEL at level 1, and move to {3,6}.

[8] Ignore 5 because 5 was selected at {3}, and back up to ϕ.

[9] Ignore 1, 2, 5, and 6 because 1, 2, 5, and 6 are adjacent to 3, FIXP at level 0.

[10] Select 4 as SEL at level 0, and move to {4}.

[11] Select 2 as FIXP at level 1, move to clique {2,4}, back up to ϕ, and stop.

Figure 3. Application of the Bron-Kerbosch Algorithm

previously ignored edges may have to be traversed. For example, if lemma 1 is applied at node {3} in Figure 3 for node SEL, when SEL is 5, the edge labeled 6 can be ignored, but only if the previously ignored edge labeled 2 is traversed. Hence, the lemma should be used selectively so that the number of new edges to be traversed is less than the number of edges to be eliminated.

The elimination of all redundant edges requires additional tests. Thus if lemma 1 is applied at node {3} for node 5, the edge labeled 2 must be traversed to generate the clique {2,3,6}, if this clique exists. However if a test reveals that nodes 2 and 6 are not adjacent, this clique cannot exist, and the edges labeled 2 and 6 can both be ignored.

These modifications were incorporated into the Bron-Kerbosch algorithm, but the performance of the algorithm was not improved because the time required to perform the additional tests was comparable to that required to traverse the redundant edges. Therefore it seems unlikely that the Bron-Kerbosch algorithm can be improved significantly.

2.2 Bron-Kerbosch Algorithm

The following formulation of the Bron-Kerbosch algorithm is very similar to Mulligan's formulation.

```
ALGORITHM_BRON-KERBOSCH: PROCEDURE;
   DECLARE S the set of all data nodes,
           NIL the empty set,
           C a global integer variable,
           COMPSUB a global set of nodes;
              /* COMPSUB is a complete subgraph containing C nodes */
   STEP_1: /* Initially COMPSUB is empty, none of the nodes have been explored,
              and all nodes are candidates which can be added to COMPSUB.
              Hence call EXTEND with arguments NIL and S. */
      C = 0;
      COMPSUB = NIL;
      CALL EXTEND(NIL,S);
   EXTEND: PROCEDURE(EXPL,CAND) RECURSIVE;
      DECLARE EXPL a local set of nodes,
                 /* EXPL = {a ε S | a adjacent to all the nodes ε COMPSUB, and
                    all cliques containing COMPSUB U {a} have been generated}.
                    Nodes in EXPL are not added to COMPSUB because this would
                    lead to cliques previously generated. */
              CAND a local set of nodes,
                 /* CAND = {a ε S | a adjacent to all nodes ε COMPSUB,
                    a ∉ EXPL}, the set of nodes called candidates that can be
                    added to COMPSUB to form new complete subgraphs. */
```

```
        NEXPL a local set of nodes,
           /* NEXPL = {a ε EXPL | a adjacent to SEL}, the new set of
              explored nodes constructed in STEP_3 of EXTEND for the
              next recursive call to EXTEND. */
        NCAND a local set of nodes,
           /* NCAND = {a ε CAND | a adjacent to SEL, a ≠ SEL}, the
              new set of candidates constructed in STEP_3 of EXTEND
              for the next recursive call to EXTEND. */
        FIXP a local variable representing one node,
           /* FIXP is the first node ε EXPL ∪ CAND adjacent to the
              most nodes ε CAND. */
        SEL a local variable representing one node;
           /* SEL is a node ε CAND selected to be added to COMPSUB. */
STEP_2: /* Choose FIXP and SEL. */
   FIXP = first node ε EXPL ∪ CAND adjacent to the most nodes ε CAND;
   IF FIXP ε EXPL
      THEN SEL = first node ε CAND not adjacent to FIXP;
      ELSE SEL = FIXP;
STEP_3: /* Add SEL to COMPSUB, increment C, and construct NEXPL and NCAND.
           Note that the number of candidates decreases for each call to
           EXTEND; hence EXTEND always returns. */
   NEXPL = {a ε EXPL | a adjacent to SEL};
   NCAND = {a ε CAND | a adjacent to SEL, a ≠ SEL};
   COMPSUB = COMPSUB ∪ {SEL};
   CAND = CAND - {SEL};
```

```
      EXPL = EXPL ∪ {SEL};
      C = C + 1;
   STEP_4: /* If NCAND and NEXPL are empty, a clique has been generated. */
      IF (NEXPL = NIL) & (NCAND = NIL) THEN print the clique of C nodes
         contained in COMPSUB;
   STEP_5: /* If NCAND is not empty, COMPSUB can be extended further. */
      IF NCAND ¬= NIL THEN CALL EXTEND(NEXPL,NCAND);
   STEP_6: /* Either NEXPL is not empty and NCAND is empty which implies
               that a previously generated clique is being constructed, or
               a new clique has been printed, or a successful return from
               EXTEND has occurred.  Hence back up by removing SEL from
               COMPSUB.  If possible, select a new SEL and attempt to
               generate more cliques.  Otherwise return. */
      C = C - 1;
      COMPSUB = COMPSUB - {SEL};
      IF there are nodes ε CAND not adjacent to FIXP
         THEN select the first such node as SEL and go to STEP_3;
         ELSE RETURN;
   END EXTEND;
END ALGORITHM_BRON-KERBOSCH;
```

2.3 Moon-Moser Graphs

Bron and Kerbosch tested their algorithm on the Moon-Moser graphs [3] which contain more cliques per node than any other graphs. These graphs have 3k nodes grouped into k triplets, and each node is adjacent to every

other node except the two nodes in the same triplet. The graph with 3k nodes contains 3^k cliques. They found that their algorithm constructed all the cliques in the Moon-Moser graph with 3k nodes in time of $O(3.14^k)$. We can gain some insight into this result by counting the number of comparisons required to generate the cliques.

For a Moon-Moser graph with 3k nodes grouped into the triplets {1,2,3}, {4,5,6}, ..., {3k-2, 3k-1, 3k}, the number of comparisons, c_k, is as follows.

Operation	Comparisons
Find FIXP	3k(3k-1)
Construct lists EXPL,CAND	3k-1
Call EXTEND	c_{k-1}
Find next SEL	1
Construct lists NEXPL,NCAND	3k-1
Call EXTEND	c_{k-1}
Find next SEL	1
Construct lists NEXPL,NCAND	3k-1
Call EXTEND	c_{k-1}

This is the best case because the choices for SEL are 2 and 3, and finding the next SEL requires only one comparison. Summing these counts yields

$$C_k = 3C_{k-1} + 9k^2 + 6k-1 .$$

This linear difference equation is easily solved giving

$$C_k = 3^k \sum_{i=1}^{k} (9i^2 + 6i-1)3^{-i} .$$

The worst case occurs when the choices for SEL are 3k-1 and 3k. In this case

$$C_k = 3^k \sum_{i=1}^{k} (9i^2 + 12i-7)3^{-i} .$$

As $k \to \infty$, both sums converge yielding

$$C_k^{best} \sim 3^{k+2.6053},$$

$$C_k^{worst} \sim 3^{k+2.6801}.$$

If the number of comparisons is an adequate measure of the time required by the algorithm, then the Bron-Kerbosch algorithm operates at the theoretical limit of $O(3^k)$.

3. IMPLEMENTATION

In most areas where clustering techniques are applied, large amounts of data are generally processed. Clique generation is often used as an important first step in classifying the data. To make clique generation practical for large data sets, it is essential to develop an efficient implementation of the fastest available algorithm, the Bron-Kerbosch algorithm.

Bron and Kerbosch implemented their algorithm in Algol, while Mulligan implemented it in PL/I. Since their implementations are identical, we will restrict our attention to Mulligan's.

Mulligan's implementation is fairly fast. However, since enormous amounts of time are generally required to process large graphs, even a modest improvement in performance is of practical significance. In his implementation, EXPL and CAND are concatenated into a single vector of integers with a pointer indicating the boundary between the two lists. A selected candidate is transferred from the candidate list to the explored list by exchanging it with the first node in the candidate list and incrementing the pointer by one. This data structure has certain advantages. Additions to and deletions from the lists are simple, and the determination of the list contents is trivial. However, it complicates the execution of the principal operations of finding FIXP, and constructing the lists NEXPL and NCAND. These operations must be performed serially node by node in loops. They could be speeded up if the lists were sorted, but this would require a more elaborate list structure and additions to and deletions from the lists would be more complicated.

We observed that the principal operations can be written in terms of set intersections as follows:

FIXP = first node i ε EXPL ∪ CAND for which CAND ∩ {nodes ε S adjacent to i} has maximum number of nodes,

NEXPL = EXPL ∩ {nodes ε S adjacent to SEL},

NCAND = CAND ∩ {nodes ε S adjacent to SEL}.

If the lists are represented by bit strings, then intersections of the lists can be computed rapidly using boolean operations which perform blocks of comparisons in parallel. Hence we chose to represent the lists of candidates and explored nodes and the rows of the adjacency matrix for an m-node graph by m-bit strings. This new data structure speeds up the principal operations, but it also creates new problems. The determination of the names and the number of nodes in a list, formerly trivial, now is fairly difficult. It would defeat the purpose of the new data structure to perform these operations bit by bit in a high-level language. Hence we chose to implement these operations in a low-level language. An efficient subroutine can be written to count the one bits in a bit string using the IBM/360 logical instruction "translate and test", which maps a byte into a table. Unfortunately, there is no IBM/360 instruction which extracts the locations of the one bits in a string. However, a subroutine which rapidly extracts the one bit locations can be written using a fast register to register add instruction.

We implemented the basic algorithm in PL/I. A listing of the PL/I procedure EXTEND and the Assembler subroutines is presented in the Appendix.

3.1 Numerical Results

The new implementation was compared to Mulligan's on several graphs. Unfortunately, accurate timing results could not be obtained because of the

local multiprogramming environment.

Some typical results are shown in Tables 1 and 2. Table 1 presents results for the Moon-Moser graphs with 3k nodes. Since the time estimates are contaminated with random errors, least squares approximations of the form ak + b were fitted to the logarithms of the times. These approximations indicate that the time required to generate 3^k cliques is proportional to 3.00^k for Mulligan's implementation and to 2.99^k for the new implementation. Given the timing errors, we can conclude that the actual execution time is probably proportional to 3^k.

Table 2 presents results obtained using data from a color-shape preference test for preschool children. Each child's performance is described by a characteristic vector of 72 bits. Two performances were judged to be similar if δ or more bits in the corresponding vectors matched. We analyzed an 80 node graph summarizing the data for 80 children. As the threshold δ decreases, the number of edges in the graph grows, and the number of cliques increases rapidly.

In both examples, the new implementation is superior to Mulligan's. Although the improvement in performance is relatively modest, it is significant in view of the high cost of analyzing large data sets.

k	Time (Seconds)	
	Mulligan	Present
5	1.43	.84
6	5.02	2.48
7	14.13	7.43
8	42.44	22.94
9	129.82	66.34

Table 1. Moon-Moser Graphs with 3k Nodes

Threshold δ	Number of Cliques	Time (Seconds)	
		Mulligan	Present
33	165	8.78	1.42
31	315	14.39	2.87
29	730	31.25	6.10
27	2348	95.10	20.57
25	7505	346.78	66.78

Table 2. Color-Shape Preference Test Graph

LIST OF REFERENCES

1. Augustson, J. G., and Minker, J., "An analysis of some graph theoretical cluster techniques," J. ACM 17, 571-588, 1970.

2. Bron, C., and Kerbosch, J., "Finding all cliques of an undirected graph," Comm. ACM 16, 575-577, 1973.

3. Moon, J. W., and Moser, L., "On cliques in graphs," Israel J. Math. 3, 23-28, 1965.

4. Mulligan, G. D., Algorithms for finding cliques of a graph, Technical Report 41, Department of Computer Science, University of Toronto, 1972.

5. Sibson, R., "Single-link cluster method," Comp. J. 16, 30-32, 1973.

APPENDIX

PL/I and ASSEMBLER PROGRAMS

```
EXTEND:
   PROCEDURE(CAND,EXPL,N) RECURSIVE;
   /*  RECURSIVE PROCEDURE GENERATING ALL POSSIBLE CLIQUES EXTENDED
       FROM THE PARTIAL SOLUTION IN "COMPSUB" USING "CAND"
       INITIALLY, CAND CONTAINS 1 BITS FOR ALL NODES PRESENT
                  EXPL IS THE NIL OR ZERO BIT STRING
       GLOBALLY DEFINED VARIABLES ARE:
                 PRNTFLG - BIT 1 => CLIQUES ARE TO BE PRINTED
                           BIT 0 => CLIQUES ARE NOT PRINTED JUST
                                    COUNTED IN NUMOUT
                 NUMOUT - COUNTER OF CLIQUES WHEN PRNTFLG IS BIT 0
                 CONNECTED - N DIMENSIONAL VECTOR, EACH ELEMENT IS
                             OF N BITS. ( ADJACENCY MATRIX )
                             CONNECTED(J) - BIT STRING REPRESENTING
                                            NODES ADJACENT TO J
                 VERTEX - N DIMENSIONAL VECTOR LIKE CONNECTED
                          VERTEX(J) IS OF N BITS WITH A 1 BIT IN THE
                          JTH POSITION ONLY                          */
       /*  ASSEMBLER SUBROUTINES                                     */
       /*  COUNT(NB,STR,CT) IS A SUBROUTINE THAT COUNTS UP THE
           NUMBER OF 1 BITS IN A BIT STRING:
           CT = # OF 1 BITS IN BIT STRING STR, WHERE STR IS OF LENGTH
                NB BYTES                                             */
   DCL COUNT OPTIONS(ASM)
             ENTRY(FIXED BIN(15,0),BIT(* ),FIXED BIN(15,0)),
       /*  XTRACT(NB,STR,LIST,M) IS A SUBROUTINE THAT EXTRACTS THE
           POSITIONS OF 1 BITS IN A BIT STRING
           LIST = LIST OF POSITIONS OF 1 BITS IN THE BIT STRING STR,
                  WHERE STR HAS LENGTH NB BYTES, AND
           M = NUMBER OF ELEMENTS IN LIST                            */
       XTRACT OPTIONS(ASM)
             ENTRY(FIXED BIN(15,0),BIT(* ),(*) FIXED BIN(15,0),
                   FIXED BIN(15,0)),
       CAND BIT(*),                  /*  CANDIDATES PASSED          */
       EXPL BIT(*),                  /*  EXPLORED NODES PASSED      */
       (NCAN,NEXP) BIT(N),           /*  NEW CAND, NEW EXPL         */
       NB FIXED BIN(15,0),           /*  NO. OF BYTES IN BIT STRINGS*/
       ASEL BIT(N),                  /*  LIST OF FUTURE SELECT NODES*/
       FIXP FIXED BIN(15,0),         /*  MOST CONNECTED NODE W.R.T. THE
                                         CANDIDATES                 */
       RLIST(N) FIXED BIN(15,0),     /*  RETURN LIST FROM ASM RCUTINES */
       ZERO BIT(N),                  /*  ZERO BIT STRING            */
                                     /*  CT, CNT ARE COUNTERS OF 1 BITS
                                         SEL IS THE SELECTED NODE
                                         IFL IS A FLAG
                                         OTHERS ARE INDEXING VARIABLES */
       (I,CNT,CT,IFL,IS,SEL,J,M) FIXED BIN(15,0);
```

```
   ASEL = ASEL & ¬ASEL;              /*  INITIALIZE                      */
   ZERO=ASEL;
   IFL=0;
   NB=N/8;                          /*  ASSUME N DIVISIBLE BY 8         */
   CNT=0;

   /*  GOING THRU CAND AND EXPL LISTS TO FIND THE NODE THAT IS
       CONNECTED TO THE MOST CANDIDATES                                  */
   IF ZERO=EXPL THEN GO TO SKP1;
                                    /*  INTEGER REPRESENTATION OF ALL
                                        EXPL NODES INTO RLIST           */
   CALL XTRACT(NB,EXPL,RLIST,M);
   DO I=1 TO M;                     /*  SEARCH THRU EXPL LIST           */
        J=RLIST(I);
                                    /*  COUNT CONNECTIONS TO CANDS.     */
        CALL COUNT(NB,CONNECTED(J) & CAND,CT);
        IF CT>CNT THEN              /*  FIND  MOST CONNECTED NODE       */
          DO;
             CNT=CT;      FIXP=J;     ASEL= ¬CONNECTED(J) & CAND;
          END;
   END;
SKP1: IF ZERO=CAND THEN GO TO SKP2;
   CALL XTRACT(NB,CAND,RLIST,M);    /*  EXTRACT NODES FROM CAND LIST    */
   DO I=1 TO M;                     /*  SEARCH THRU EXPL LIST           */
        J=RLIST(I);
        CALL COUNT(NB,CONNECTED(J) & CAND,CT);
        IF CT>CNT THEN
          DO;
             CNT=CT;   FIXP=J;   IFL=1;   ASEL = ¬CONNECTED(J) & CAND;
          END;
   END;
SKP2:  CALL XTRACT(NB,ASEL,RLIST,M);  IS=1;
   IF IFL=0 THEN                    /*  IFL=0 => FIXP IS IN EXPL LIST   */
     DO;                            /*  SELECTED NODE MUST BE A CAND.
                                        INSTEAD OF FIXP, CHOOSE A CAND.
                                        NOT CONNECTED TO FIXP           */
        SEL=RLIST(IS);     IS=IS+1;
     END;
   ELSE SEL=FIXP;                   /*  ELSE FIXP IS A CAND., CHOOSE
                                        FIXP                            */
```

```
  DO WHILE (IS <= M+1);              /*  WHILE THERE ARE STILL SEL'S   */
                                     /*  CONSTRUCT NEW CAND, EXPL      */
      NEXP = EXPL & CONNECTED(SEL);
      NCAN = CAND & CONNECTED(SEL) & ¬VERTEX(SEL);
      C=C+1;
      COMPSUB(C)=SEL;                /*  ADD SELECTED NODE TO PARTIAL
                                         SOLUTION OF CLIQUE            */
                                     /*  NO MORE CANDIDATES AND EXPLORED
                                         NODES => A CLIQUE IS FOUND    */
      IF (NCAN | NEXP) = ZERO THEN
           IF PRNTFLG THEN
                PUT EDIT((COMPSUB(I) DO I=1 TO C)) (SKIP,(C)F(3));
           ELSE NUMOUT=NUMOUT+1;
                                     /*  IF THERE ARE MORE CANDIDATES TO
                                         BE EXPLORED , CALL EXTEND TO GO
                                         DOWN ONE MORE LEVEL           */
      ELSE IF NCAN ¬= ZERO THEN CALL EXTEND(NCAN,NEXP,N);
      C=C-1;                         /*  DELETE EXPLORED NODE FROM
                                         CLIQUE                        */
                                     /*  DELETE EXPLD. NODE FORM CAND  */
      CAND = CAND & ¬VERTEX(SEL);
                                     /*  ADD EXPLORED NODE TO EXPL     */
      EXPL = EXPL | VERTEX(SEL);
      SEL=FLIST(IS);                 /*  SELECT ANOTHER NODE OUT OF LIST
                                         OF CANDIDATES NOT CONNECTED TO
                                         FIXP                          */
      IS=IS+1;
  END;
END EXTEND;
```

```
COUNT    BEGIN
         L     3,0(0,1)          R3=ADDR(LENGTH OF STRING IN BYTES)
         L     4,4(0,1)          R4=ADDR(STRING)
         L     5,8(0,1)          R5=ADDR(COUNT)
         LA    9,1               R9=CONSTANT 1
         LA    8,0               R8=COUNT REGISTER
         LH    6,0(0,3)          OBTAIN LENGTH OF STRING
         AR    6,4               R6=ADDR OF END OF STRING
         SR    4,9               SCAN START ONE BYTE TO THE LEFT
LOOP     LA    2,0               CLEAR R2,R1
         LA    1,0
         TRT   1(256,4),TABLE    TRANSLATE AND TEST
         CR    6,1               HAS CAN REACHED END?
         BC    12,OUT
         AR    8,2               NO.  ADD TO COUNT REGISTER
         LR    4,1               RESET SCAN ADDR
         B     LOOP              BRANCH BACK
OUT      STH   8,0(0,5)          YES.   STORE INTO COUNT
         LEAVE
TABLE    DC    X'000101020102020301020203020303040102020302030304'
         DC    X'020303040304040501020203020303040203030403040405'
         DC    X'020303040304040503040405040505060102020302030304'
         DC    X'020303040304040502030304030404050304040504050506'
         DC    X'020303040304040503040405040505060304040504050506'
         DC    X'040505060506060701020203020303040203030403040405'
         DC    X'020303040304040503040405040505060203030403040405'
         DC    X'030404050405050603040405040505060405050605060607'
         DC    X'020303040304040503040405040505060304040504050506'
         DC    X'040505060506060703040405040505060405050605060607'
         DC    X'040505060506060705060607060707 08'
         END
```

```
XTRACT   BEGIN
         L     2,0(0,1)        R2=ADDR OF N, # OF BYTES
         L     3,4(0,1)        R3=ADDR OF STRING
         L     4,8(0,1)        R4=ADDR OF PASSED LIST
         LA    9,8             R9=CONSTANT 8
         LA    6,1             R6=CONSTANT 1
         LA    5,0             R5=BYTE COUNT
         LA    8,0             R8=BIT COUNT
         LA    10,2            R10=CONSTANT 2
LOP      IC    7,0(0,3)        PUT ONE BYTE IN R7
         SLL   7,24            SHIFT LEFT
LOP1     AR    8,6             UP BIT COUNTER
         ALR   7,7             SHIFT LEFT ONE
         BC    12,NOAD         CARRY?     NO, GO TO NOAD
         STH   8,0(0,4)        YES. STORE BIT INDEX
         BC    2,NXT           ZERO?  YES, NEXT BYTE
         AR    4,10            READY FOR NEXT STORE
         B     LOP1            BACK FOR ANOTHER SHIFT
NXT      AR    4,10            READY FOR NEXT STORE
         B     NEXT
NOAD     BC    4,LOP1          NO CARRY. NOT ZERO. SHIFT AGAIN
NEXT     AR    5,6             UP COUNT OF BYTES
         LR    8,5
         SLA   8,3             RESET BIT COUNTER
         AR    3,6             NEXT BYTE ON STRING
         CH    5,0(0,2)        # OF BYTES EXCEED LENGTH?
         BC    4,LOP           NO. GET ANOTHER BYTE
         L     7,12(0,1)       R7 = ADDR OF FOURTH PARAMETER
         S     4,8(0,1)
         SRA   4,1             COUNT # OF INDEX STORED
         STH   4,0(0,7)        STORE AND RETURN
         LEAVE
```

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-655	2.	3. Recipient's Accession No.
4. Title and Subtitle CLUSTERING BY CLIQUE GENERATION			5. Report Date June, 1974 6.
7. Author(s) CHIH-MENG CHENG			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

17. Key Words and Document Analysis. 17a. Descriptors

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Unlimited	UNCLASSIFIED	30
	20. Security Class (This Page) UNCLASSIFIED	22. Price